Data Structure Design Guidelines

R. C. Tausworthe
DSN Data Systems Development Section

Proper modularization of software designs is more than mere segmentation of a program into subfunctions as dictated by control-logic topologies, as might be suggested by classical structured programming, wherein a limited number of program control-logic structures are permitted. Analyzing data connectivity between program segments can be far more complex than analyzing control flow, unless conscientious precautions are taken to avert this possibility. For this reason, data connectivity design should adhere to a discipline which minimizes both data and control-flow connections. This article discusses such considerations within a top-down, hierarchic, structured-programming approach to software design.

I. Introduction

As I shall be dealing with it, design is meant to be that activity which defines program data structures and logical algorithms in response to, and conforming with, a software functional specification. It consists of program organization, data manipulations, input/output (I/O) procedures, and the like, carried to a level of detail sufficient to serve as the working basis for coding and operational implementation. The basic elements required to effect a good program design are an understanding of the function to be served and the mechanisms available to carry out the job.

The data structure design guidelines I shall describe are prompted by what I call "top-down, modular, hierarchic, structured development of software." In doing a top-down, modular, hierarchic, structured design, one starts with an end-to-end overall definition of the program and analyzes it into a number of component parts according to a set of decomposition rules. In terms of flowcharts, one starts with a single box that represents the entire program at the top hierarchic level, and expands that box into a flowchart at the next level, which displays the component subfunctions as a structured algorithm, in keeping with certain flowchart-topology rules.

Each of the subfunctions is given a precise end-to-end subspecification, some of which will be expanded into separate flowcharts at the next design level, and so on, until the final collection of subspecifications can be coded directly, without functional ambiguity. The American National Standards Institute (ANSI) standard (Ref. 1) technique for depicting those submodules which are to be expanded by subsequent flowcharts is by "striping" that flowchart symbol on the parent flowchart. I will, therefore, refer to such submodules as striped submodules They are also referred to by others (Ref. 2) as stubs.

Such hierarchic decomposition identifies the programming process as a step-by-step decomposition of mathematical functions into structures of logical connectives and subfunctions which ultimately can be realized directly in the programming language to be used. Such a decomposition tends to channel detail into functional levels which aid human comprehension, and thereby, provides a way to control complexity in a disciplined, systematic way.

Certain flowchart topologies, or logical connectivities of the subfunctions, limited to iterations and nestings of a canonic-structured set (Ref. 2) have been shown (Ref. 3) to produce programs that are readable, understandable, codable, testable, maintainable, modifiable, and manageable. Control branching is entirely standardized so that the flowchart, accompanying narrative, and resultant code can be read from top to bottom without having to trace the branching logic in any intricate, convoluted way.

But proper modularization of software is more than just segmentation of a program into subfunctions as dictated by control-logic topologies. One may conceivably erase all the control flow lines from a flowchart and replace them by lines representing the data accesses instead, as a graphic way to identify operations on the data and to display data interconnectivity between executing modules. Such a chart would undoubtedly be convincing evidence that analyzing data connectivity can be far more complex than analyzing program control flow, unless conscious precautions are taken to avert this possibility.

For this reason, data connectivity design should, from the very first, be made to adhere to a discipline which minimizes module connections and organizes it into understandable units. Such a discipline, when coupled with structured control-logic design methods, offers the possibility of maintaining program clarity and correctness in both data flow and control flow.

II. Information, Data, and Storage Structures

A program operates on data. An information structure is a representation of the elements of a problem or of an applicable solution procedure for the problem; a data structure is a representation of the ordering and accessibility relationships among data items without regard to storage or implementation considerations; and a storage structure is a representation of the logical accessibility between data items as stored in a computer (Ref. 4). For example, in the vector-algebra problem $Ax = b$, the vectors x and b and the matrix A are information structures; when we agree to represent this problem in the form of dimensioned arrays A(N,N), X(N), B(N), then A, B, and X become data structures; when we represent these in computer memory, as for example, by the mapping

$$\mathit{location}(\mathrm{A}[\mathrm{I},\mathrm{J}]) = \mathit{location}(\mathrm{A}[1,1]) + N^{*}(\mathrm{I}-1) + \mathrm{J} - 1$$

then this becomes the storage structure.

A data structure is generally specified as a set of data items (variables or constants), each typed (a) by a range of values (such as logical, integer, real, complex, double-precision, character, string, or an enumerated set of values) and (b) by a connectivity of items within the structure (such as are implicit in a linear list, stack, queue, deque, orthogonal array, tree, ring, or graph). Perhaps the simplest example of a data structure is a single integer-valued variable.

The data structures which one is apt to use most often depend on the facility with which the programming language to be used accommodates that structure. For example, FORTRAN accommodates integer, real, and complex data types in simple or array data structures. It is certainly possible in FORTRAN to create and manipulate a queue of string records as a data structure; but it is not as easy as it is in, for example, PL/1, where string variables and linked-list data structures are within the language repertoire.

A data structure also possesses another attribute having to do with when and where it is accessed in the program. This is its scope of activity (or merely, its scope). The scope of a structure extends from the earliest point in a program where information appears in that structure, until the latest point that structure is needed, either by the current module, or by another interfacing subsequent module. The structure is active whenever the program is executing within the scope of that structure. The scope need not be continuous. For example, an index variable for an iteration only is active during the iteration, and may be reused by other parts of a program once the iteration has been completed.

III. Data Structure Hierarchies

Dijkstra (Ref. 5) formulated the solution of a programming problem in terms of a set of "levels of abstraction," or concepts capable of being implemented (and interpreted) in many ways, but which were perhaps not fully understood at any particular stage of development. Later stages then provided refinement to each concept until the program was entirely complete. The use of abstractions provided a mechanism for hierarchic refinement by which it was possible to express those details that were known and relevant at a particular time, and to defer for later refinement, those details which were not.

Hoare (Ref. 6) characterizes an abstract resource, such as a data structure, by three sets of such hierarchics: (1) the representation of the abstract resource, or a set of symbols which one may substitute for the physical aspects of the actual resource; (2) a set of manipulations which provide the transformation rules for representations as a means of predicting the effect of similar manipulations on the physical resources; and (3) a set of axioms which state the relationship and extent to which the physical properties of a resource are shared by their computer representation. The extent to which an abstraction leads to a successful program depends on the extent to which (a) the axioms describe the problem, (b) the axioms model the program behavior, and (c) the choice of a representation, with its manipulations, yields acceptable performance merits.

The way abstractions are formulated also greatly influences the extent and likelihood that a program will need major revision during the development process. This is the case because the nature of the data and the processing they require tend to influence the data structure design significantly. Premature representation of a data structure during design, when the needs of the structure are relatively unknown, leads to errors in judgement that may go undetected until too late for effective removal. The use of abstractions during design can postpone some of the decisions on data representation until a more appropriate time in the development.

IV. Levels of Access

Data structures to be used in a program are particularly well suited (Ref. 6) to being designed into levels of abstraction imposed by the hierarchic decomposition of program specifications. In the top-down method, the top-layer considerations are concerned with the problem, and deeper layers traverse the span to programming language. The specification hierarchy for a data structure will thus begin with one fitting the needs of the problem and wind up with detail at the programming language level.

For example, suppose, in the upper layers of the design, that a module function may recognize the need for a "stack" to hold certain data. No more information is supplied at that level, not even the name, because no other interfaces appear. However, at some eventual hierarchic detailing of the module, the name will become important, as well as perhaps those functions which fetch and store data in the stack. Upon hierarchic expansion of these functions, more detail is needed about the stack, such as its size and the pointer to its top element. Eventually, the entire detail of the stack, down to the bit-by-bit machine configuration, will be specified in one form or another.

The hierarchy of definition thus describes the data structure in *levels of access*. At the top, the only access is through a vague notion of the data to be held; at deeper levels, the structure is accessed by name, then by increasingly more detailed operations, until, at the final level, the individual components are accessible. A level of access for a set of resources is defined as an interface through which all accesses to any constituent part of a resource must pass, except for those at deeper levels within the hierarchy.

Extending the example above, let us suppose that data at some level can be accessed in a stack by way of operations PUSH and PULL. Then let all accesses to the stack in the rest of the program, except for accesses within the access functions themselves, be made only via this level of access. Accesses to stack components within the PUSH-PULL functions have a deeper, more detailed level of access to the data structure. Then the access functions *own* the structure at each level of access.

The concept may be extended; suppose functions PUSH(*stack*) and PULL(*stack*) represent a level of access for a set of stack structures whose names can be substituted for the syntactic variable *stack* above. Again, the access functions own the set of stacks exclusively at that level of access in the sense that modules outside PUSH and PULL wishing to access a stack *must* do so only through these functions.

The general idea here is that a data structure (and, indeed, any resource) may be characterized by its levels of access as well as by the function it serves. Levels of access, then, can provide a conceptual framework for achieving a clear and logical design. At the lowest level are the access functions for individual resource units, such as arithmetic registers, memory cells, file elements, etc. File elements are built into records by defining functions to process groups of file elements as a unit; records are built into files by defining functions to process groups of records as a unit; and so on, up the hierarchy. Each level supports an important abstraction of the hierarchic buildup of the resource.

Each access level consists of one or more externally accessible functions which share commonly owned resources. The connections in control and data among the various access modules induced by the top-down hierarchy are then limited in a natural way. Every resource used by a program will eventually be represented in a hierarchy whose levels map the needs of the problem into characteristics of the resource.

V. Data Design

As was indicated earlier, data-flow analysis is a natural tool for specifying what a program function is in terms of transformations of input data to the output wanted. In a design, which specifies how the computer is to implement these, it is useful to identify module interfaces to show the precedence of data creation and use among modules, and to promote understanding of the program interactions. For example, if data created in modules A and B are going to be further processed by module C, then the execution of A and B must precede C; if A and B do not share data, either may be executed first.

Data-flow diagrams depict the activity of a program module as reading certain input data structures and writing other output data structures according to predefined rules. Such diagrams can be every bit as useful as flowcharts, because they provide a means of attacking a problem in which questions of control, which at the early stages of design only tend to obscure the solution anyway, are secondary. They further provide a means to identify, and then to minimize, data-connections and side-effects among modules. They fit in with the top-down, hierarchic, modular, structured design discipline. They are eminently suitable as documentation to communicate the overall program organization. They identify the elements most important to the program mainstream, so that priorities and alternate operational modes can be established. In summary, data connectivity diagrams (data-flow charts), with their accompanying explanatory narrative, form another effective tool for the designer's bag.

Probably the most effective use of data-connection analysis will occur at the highest levels of the design. Then, as design progresses, data interconnectivity becomes more firmly established in the mind of the designer (and any reviewers), so graphic aids diminish in value. This is just the opposite of flowcharting, where the control at the top levels tends to be rather non-contributory to understanding, but becomes exceedingly more important at the deeper levels.

The data-connection guideline is the following: Design the control logic of a module so as to be independent of the way the data are structured whenever practicable, and modularize accesses to data structures so that if data are restructured at a later time (e.g., for more efficiency), only the access functions need be altered; organize submodules to minimize data interfaces whenever possible.

VI. Data Structure Design

Data structuring is primarily concerned with selection of type. Each programming language has certain elementary (unstructured) types, such as integers and reals, which form the basis of more extended, or structured, types. Then each new data structure typed is defined in terms of previously defined types, and there is a corresponding set of operations valid on that type.

The fundamental aspects of data structure design are: (1) deciding when to save data rather than regenerate them from the input, and (2) deciding how to store them when they are to be saved. Such decisions not only depend on the input data (type) but on the amount (e.g., to store in files versus core), their characteristics (e.g., sparse versus dense within the information structure), and the uses to which such data are to be put (e.g., predominance of comparisons versus updates). One important decision is the degree of packing to save space versus the lack of packing to save execution time. Other decisions have to be made concerning whether the data accesses are to be direct (i.e., accessed directly within the structure) or indirect (i.e., accessed indirectly through a surrogate structure of pointers).

The principal key to making such decisions is experience. No generalized guidelines can relate what data structure best fits the needs of specific problem. However, hierarchic abstraction does provide a generalized procedure for linking experience and expertise to the needs of the problem.

VII. Documentation of Data Structure Design

Another key toward effecting a good data-structure design, as well as promoting correctness in programming, is worthwhile documentation of the data structure. Such documentation can be organized in the same hierarchic levels of detail as emerged naturally in the design process. In fact, if the designer sets down the data design in this form from the beginning and maintains it throughout during the design process, then the documentation forms the vehicle for design.

Such documentation keeps track of the current state of the program requirements and all assumptions concerning its data structures, their levels of access, etc., up to the current phase. Moreover, programmers should be encouraged to make this hierarchic, top-down, concurrent documentation record not only the formal, definitive aspects of a structure, such as how the structure is formed and what its levels of access are, but also the more informal descriptive aspects of the problem, such as the rationale why the structure is defined the way it is.

The rationale of a program and its data structures is for the benefit of humans, not the computer. If this rationale is based on the hierarchic structuring of detail into increasingly refined levels of access, then humans can comprehend program complexity at each level by regarding the next lower level as a functional subunit.

Documentation also forms the basis for the assessment of program correctness, whether it be by formal proof, informal desk-checking, or testing the running program. In the next Section, I give guidelines relating to the level and content of documentation for data structures.

VIII. Data Structure Documentation Guidelines

The overall guideline which has governed the remainder of this section is the following: Documentation of each program submodule should exist to a sufficient degree that correctness can be assessed rigorously on the basis of its control logic and auditably for functional completeness.

To this end, I shall assume that the control logic for a given submodule has been specified completely (as structured programming does), so that module control is explicit, and therefore fulfills the guideline. For all decisions to be explicit and determinable within an individual submodule with no other aid than references to preceding levels of the design, the unstriped (non-stub) subfunctions must give explicit settings to all control flag assignments. A striped (stub) subfunction at the current level which is specified to alter a control flag at a later level, but used at the current or prior level, must be accompanied by documentation which details explicit flag settings and the rationale for the setting.

Data structures accessed by unstriped submodules must be declared as to specific type attributes necessary for the intended programming language to access that structure without any ambiguity. Internal data structures accessed by striped modules, not pertinent to control logic or functional correctness, as specified above, may be detailed in later levels in the design. Further expansions of striped boxes successively provide more and more detail about the data structures and requirements involved. Specifically, each further detailing of a data structure definition must be made consistent with every previous assumption concerning its use as a minimum. The final, explicit form of a data structure definition should contain: (i) the structure name; (ii) its mnemonic derivation; (iii) type attributes (e.g., real, string/array variable, simple variable, etc.); (iv) range of values; (v) scope of activity (i.e., over what portions of the program the structure is not available for reassignment or reuse by other parts of the program); (vi) description of the use of the data structure in the program; and (vii) a list of any data structures which share storage with this structure.

Declaration, or declaration and initialization of a *new* data structure may appear as an entry requirement of the current submodule, to be performed in specified, previously defined modules. Such actions are documented by annotations to the flowchart and code for the current module; the actual declaration/initialization code is located within the specified modules (striped or unstriped), indented (if permitted by the programming language) to show that it is a later addition to that module (not contributing to nor detracting from the previous assessment of correctness) and annotated to indicate the later module which requires this initialization.

Data Structures may be referenced by striped modules in generic terms when not related to control-logic correctness. Assumptions made in such references must be consistent with the current state of the data structure definition. For example, a striped module may state that a set of characters is "put in the name table," whereas the unstriped submodule which implements that function must be specific, as "NPTR=NPTR+1, NAME(NPTR)=N$", in which NPTR, NAME, and N$ appear as appropriate detailed declarations in a Data Structure Definition Table.

A data structure referred to in generic terms, or any other way other than by its specific name, should have an entry in the software design document glossary, which then gives the actual structure name.

The current state of every data structure definition should be maintained in a Data Structure Definition Table in the software design document. This table either contains the definition or gives an explicit reference to defining material elsewhere in the design document. This table can be listed in alphabetic order for ease in locating structures referred to.

For readability, it is useful to provide the mnemonic derivation of all data-structure names used in a submodule unless such names have previously occurred in a direct ancestrial module. Names appearing in "cousin" submodules, unreferenced in their common ancestor should repeat the mnemonic derivation for ease in reading.

IX. Summary

The guidelines for data structure design are very highly influenced by the control-logic design and other design aspects of a program. I have indicated in this paper how the top-down, hierarchic, modular, structured-program approach is particularly well suited to effective data structure design because it permits the postponement of data-structure decisions until the requirements for a particular data representation are more concrete, when more is known about the program behavior and the characteristics of the data. Errors in judgement tend thus to be averted and easier to correct when detected.

References

1. *American National Standard Flowchart Symbols and Their Usage in Information Processing*, ANSI X3.5-1970, American National Standards Institute, Inc., Sept. 1, 1970.
2. Mills, H. D., *Mathematical Foundations of Structured Programming*, IBM Document FSC72-6012, Federal Systems Division, IBM Corp., Gaithersburg, Md., February 1972.
3. Baker, F. T., and Mills, H. D., "Chief Programmer Teams," *Datamation*, Vol. 19, No. 12, pp. 38–61, December 1973.
4. Robert, D. C., "File Organization Techniques," *Advances in Computers*, Vol. 12, Academic Press, Inc., New York, 1972.
5. Dijkstra, E. W., "Notes on Structured Programming," in *Structured Programming*, pp. 1–82, Academic Press, Inc., New York, 1972.
6. Hoare, C. A. R., "Notes on Data Structuring," in *Structured Programming*, pp. 83–174, Academic Press, Inc., New York, 1972.